# 多线程并发编程分享

# 概念

进程与线程

* 现在的操作系统都是多任务操作系统，允许多个进程在同一个CPU上运行。
* 每个进程都有独立的代码和数据空间，称为进程上下文
* CPU从一个进程切换到另一个进程所做的动作被成为上下文切换，通过  
  频繁的上下文切换来让这些进程看起来像是在同时运行一样
* 进程的运行需要较多的资源，操作系统能够同时运行的进城数量有限，并且进程间的切换和通信也存在较大开销。
* 为了能并并行的执行更多的任务，提升系统效率，才引入了线程概念。
* 线程是CPU调度的最小单位，是进程的一部分，只能由进程创建，共享进程的资源和代码
* 以Java进程为例，它至少有一个主线程(main方法所在的线程)，通过主线程可以创建更多的用户线程或者守护线程，线程可以有自己独享的数据空间，同时线程间也共享进程的数据空间

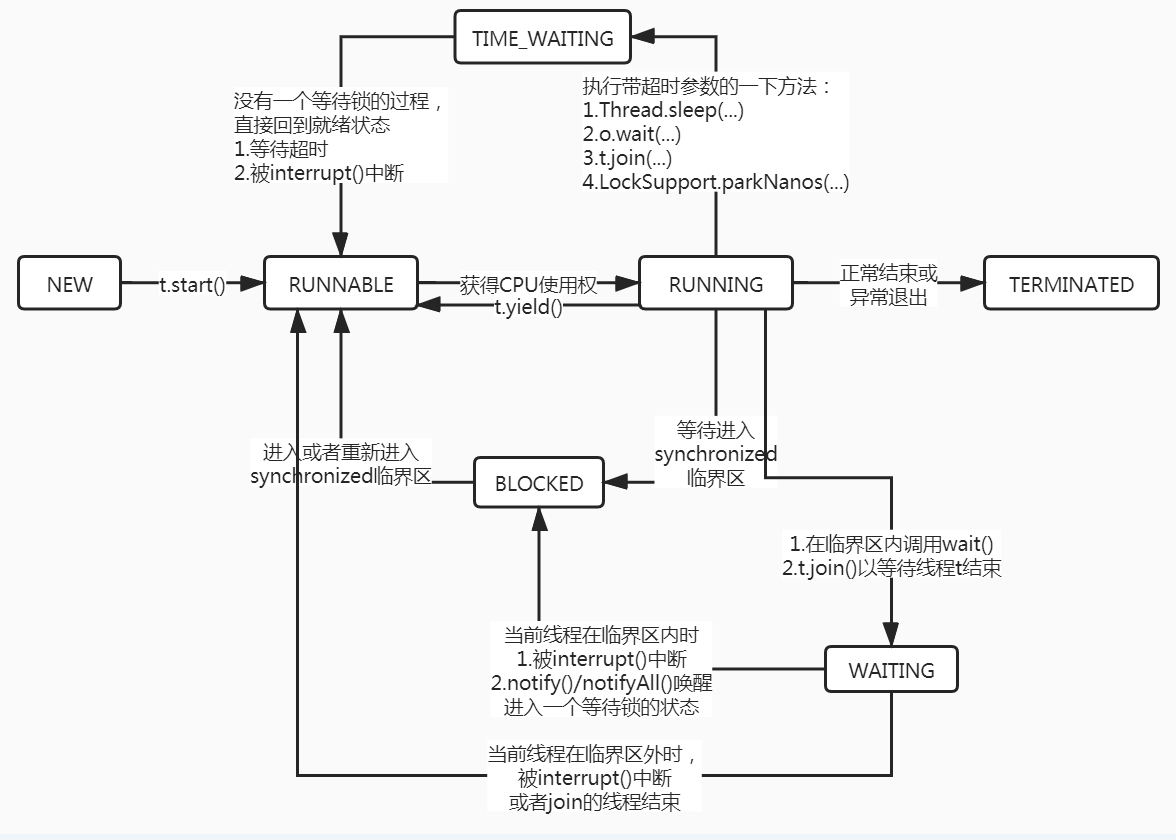
并发与并行

* 并行的概念：如果一个CPU有多个核心，并允许多个线程在不同的核心上同时执行，称为“多核并行”，这里强调的是同时执行。
* 并发的概念：比如在单个CPU上，通过一定的“调度算法”，把CPU运行时间划分成若干个时间片，再将时间片分配给各个线程执行，在一个时间片的线程代码运行时，其它线程处于挂起等待的状态，只不过CPU在做这些事情的时候非常地快速，因此让多个任务看起来“像是”同时在执行，本质上同一时刻，CPU只能执行一个任务。

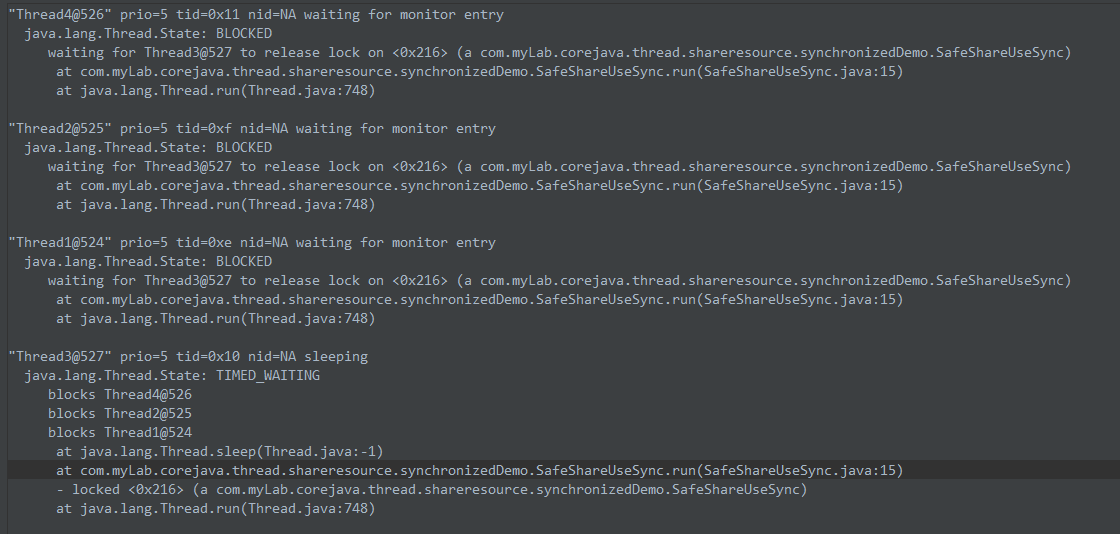
线程状态&状态间转换

* 新建NEW：线程被新创建时的状态，在堆区中被分配了内存
* 就绪RUNNABLE&READY：线程调用了它的start()方法，该线程进入就绪状态，虚拟机会为其创建方法调用栈和程序计数器，等待获得CPU的使用权
* 运行RUNNING：线程获取了CPU的使用权，执行程序代码，只有就绪状态才有机会转到运行状态
* 阻塞BLOCKED：位于对象锁池的状态，线程为了等待某个对象的锁，而暂时放弃CPU的使用权，且不参与CPU使用权的竞争。直到获得锁，该线程才重新回到就绪状态，重新参与CPU竞争，这涉及到“线程同步”
* 等待WAITING：位于对象等待池的状态，线程放弃CPU也放弃了锁，这涉及到“线程通信”
* 计时等待TIME\_WAITING：超时等待的状态，它会放弃CPU但是不会放弃对象锁
* 终止TERMINATED&DEAD：代码执行完毕、执行过程中出现异常、受到外界干预而中断执行，这些情况都可以使线程终止

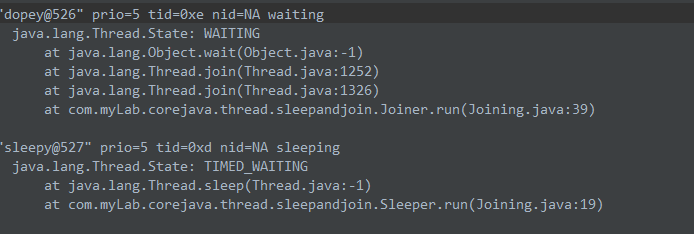
线程状态间转换图



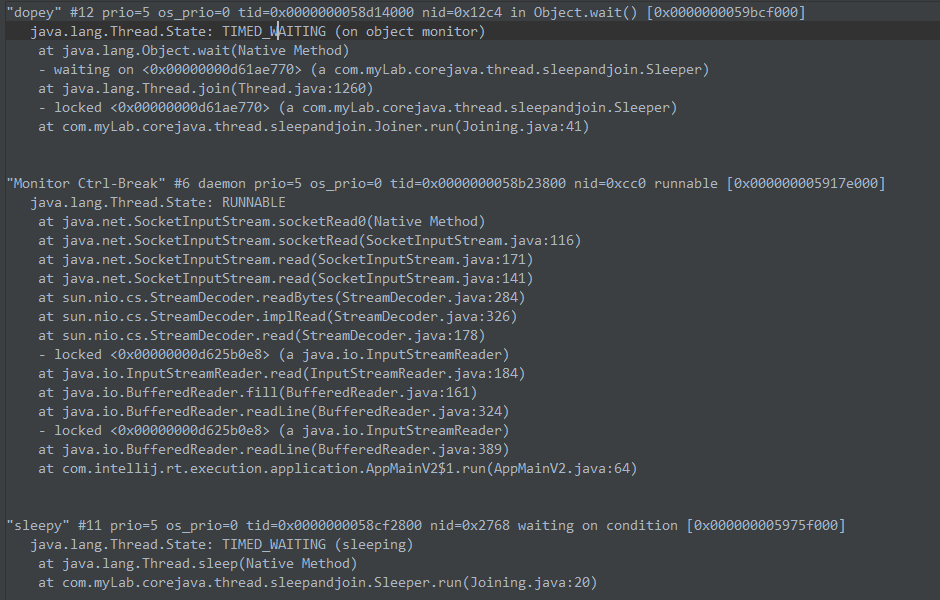
Thread3持有对象锁，Thread1,2,4进入等待获取锁时的状态是BLOCKED



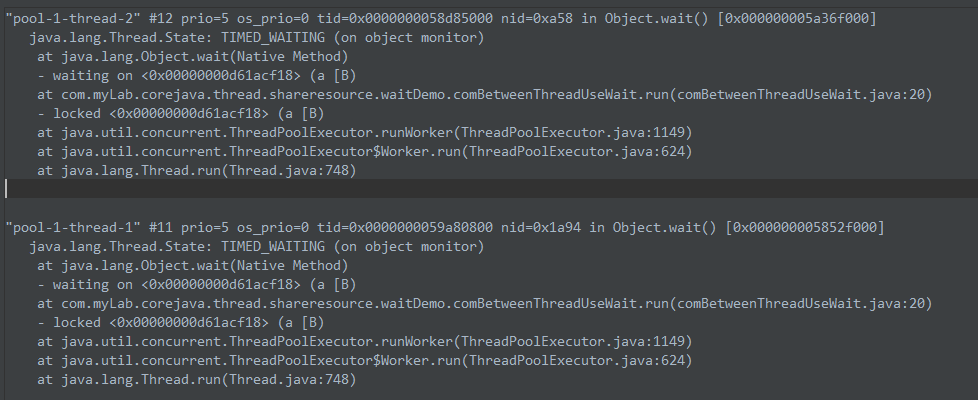
dopey线程调用sleepy.join()后，dopey线程处于WAITING状态，会等待sleepy线程结束，sleepy线程由于调用了sleep()方法，处于TIMED\_WAITING状态



如果dopey线程调用sleepy.join(…)方法，dopey会进入TIMED\_WAITING状态，它会在超时时间内等待sleepy线程结束，如果超时了sleepy线程还未结束，dopey不会继续等待，它会继续运行



调用了wait(…)方法之后会进入TIMED\_WAITING状态，超时等待



java对于线程的编程支持

* Thread类常用方法
  + t.start() 启动线程t，线程状态有NEW变为RUNNABLE，开始参与CPU竞争
  + t.checkAccess() 检查当前线程是否有权限访问线程t
  + t.isInterrupted() 检查线程t是否要求被中断
  + t.setPriority() 设置线程优先级：1-10，值越大，得到执行的机会越高，一般比较少用
  + t.setDaemon(true) 设置线程为后台线程，代码演示1
  + t.isAlive() 判断线程t是否存活
  + t.join()/t.join(1000L) 当前线程挂起，等待t线程结束或者超时，代码演示2
  + Thread.yield() 让出CPU，如果有锁，不会让出锁。转为RUNNABLE状态，重新参与CPU的竞争
  + Thread.sleep(1000L) 让出CPU，不让锁，睡眠1秒钟之后转为RUNNABLE状态，重新参与CPU竞争
  + Thread.currentThread() 获取当前线程实例
  + Thread.interrupt() 给当前线程发送中断信号
* wait和sleep的差异和共同点，代码演示3
  + wait方法是Object类的方法，是线程间通信的重要手段之一，它必须在synchronized同步块中使用；sleep方法是Thread类的静态方法，可以随时使用
  + wait方法会释放synchronized锁，而sleep方法则不会
  + 由wait方法形成的阻塞，可以通过针对同一个synchronized锁作用域调用notify/notifyAll来唤醒，而sleep方法无法被唤醒，只能定时醒来或被interrupt方法中断
  + 共同点1：两者都可以让程序阻塞指定的毫秒数
  + 共同点2：都可以通过interrupt方法打断
* sleep与yield，代码示例4
  + 线程调用sleep方法后，会进入TIMED\_WAITING状态，在醒来之后会进入RUNNABLE状态，而调用yield方法后，则是直接进入RUNNABLE状态再次竞争CPU
  + 线程调用sleep方法后，其他线程无论优先级高低，都有机会运行；而执行yield方法后，只会给那些相同或者更高优先级的线程运行的机会
  + sleep方法需要声明InterruptedException，yield方法没有声明任何异常。

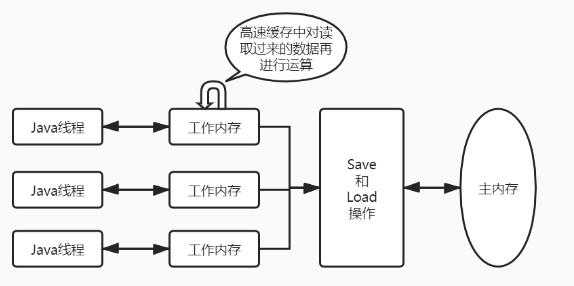
线程池

线程的创建和销毁会消耗资源，在大量并发的情况下，频繁地创建和销毁线程会严重降低系统的性能。因此，通常需要预先创建多个线程，并集中管理起来，形成一个线程池，用的时候拿来用，用完放回去。

* 常用线程池：FixedThreadPool，CachedThreadPool，ScheduledThreadPool，代码演示5
* 主要关注的功能：shutDown方法；shutDownNow方法；execute(Runnable)向线程池提交一个任务，不需要返回结果；submit(task)向线程池提交一个任务，且需要返回结果，这里涉及到Future编程模型，代码演示6

线程安全

* 怎么理解线程安全？线程安全，本质上是指“共享资源”在多线程环境下的安全，不会因为多个线程并发的修改而出现数据破坏，丢失更新，死锁等问题。
* 为什么会出现线程不安全？个人的一些思考，读操作是线程安全的，它不会改变值；写操作也是线程安全的，这里的写操作是指对于内存或者硬盘上的值进行更改的那个动作，这个动作本身是具有原子性的。有很多人说，共享资源不安全是因为“并发的写”，这里我想说“写”这个动作本身不会破坏资源的安全性。这里要结合操作系统的工作特点来说明一下这个问题。



各个线程从主内存中读取数据到工作内存中，然后在工作内存中根据代码指令对数据进行运算加工，最后写回主内存中。

* 引申出线程安全要解决的三个问题
  + 原子性，某个线程对共享资源的一系列操作，不可被其他线程中断和干扰。
  + 可见性，当多个线程并发的读写某个共享资源时，每个线程总是能读取到该共享资源的最新数据。

举例：

//线程1执行的代码

int i = 0;

i = 10;

//线程2执行的代码

j = i;

假若执行线程1的是CPU1，执行线程2的是CPU2。由上面的分析可知，当线程1执行 i =10这句时，会先把i的初始值加载到CPU1的高速缓存中，然后赋值为10，那么在CPU1的高速缓存当中i的值变为10了，却没有立即写入到主存当中。

此时线程2执行 j = i，它会先去主存读取i的值并加载到CPU2的缓存当中，注意此时内存当中i的值还是0，那么就会使得j的值为0，而不是10.这就是可见性问题，线程1对变量i修改了之后，线程2没有立即看到线程1修改的值。

* + 有序性，单个线程内的操作必须是有序的。

解释一下什么是指令重排序，一般来说，处理器为了提高程序运行效率，可能会对输入代码进行优化，它不保证程序中各个语句的执行先后顺序同代码中的顺序一致，但是它会保证程序最终执行结果和代码顺序执行的结果是一致的。

int a = 10; //语句1

int r = 2； //语句2

a = a + 3; //语句3

r = a\*a; //语句4

比如上面的代码中，语句1和语句2谁先执行对最终的程序结果并没有影响，那么就有可能在执行过程中，语句2先执行而语句1后执行。

但是执行顺序不可能是 语句2—语句1—语句4—语句3，因为这样会改变最终结果。

虽然重排序不会影响单个线程内程序执行的结果，但是多线程呢？下面看一个例子：

//线程1:

context = loadContext();   //语句1

inited = true;             //语句2

//线程2:

while(!inited ){

sleep()

}

doSomethingwithconfig(context);

上面这段代码在单线程看来，语句1和语句2没有必然联系，那如果这时发生了指令重排序，语句2先执行，那这时线程2会认为初始化已经完成，直接跳出循环，但其实线程1的初始化不一定完成了，这样就会产生程序错误。

线程同步

线程同步指的是线程之间的协调和配合，是多线程环境下解决线程安全和效率的关键。主要包括四种常用方式来实现

* 临界区，表示同一时刻只允许一个线程执行的“代码块”被称为临界区，要想进入临界区则必须持有锁
* 互斥量，即我们理解的锁，只有拥有锁的线程才被允许访问共享资源
* 自旋锁：与互斥量类似，它不是通过休眠使进程阻塞，而是在获取锁之前一直处于忙等(自旋)阻塞状态。用在以下情况：锁持有的时间短，而且线程并不希望在重新调度上花太多的成本，"原地打转"。
* 信号量，允许有限数量的线程在同一时刻访问统一资源，当访问线程达到上限时，其他试图访问的线程将被阻塞
* 事件，通过发送“通知”的方式来实现线程的同步

Java中对实现线程安全与线程同步提供哪些主要的能力

* Volatile，被volatile修饰之后就具备了两层语义：1. 保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。2. 禁止进行指令重排序，即对一个变量的写操作先行发生于后面对这个变量的读操作

看下面一段代码：

//线程1

boolean stop = false;

while(!stop){

doSomething();

}

//线程2

stop = true;

这段代码是一种典型的多线程写法，线程1根据布尔值stop的值来决定是否跳出循环；而线程2则会决定是否将布尔值stop置为true。如果线程2改变了stop的值，但是却迟迟没有写入到主存中，那线程1其实还以为stop=false，会一直循环下去。但是用volatile修饰之后就变得不一样了：

　　第一：使用volatile关键字会强制将修改的值立即写入主存；

　　第二：使用volatile关键字的话，当线程2进行修改时，会导致线程1的工作内存中缓存变量stop的缓存行无效（反映到硬件层的话，就是CPU的L1或者L2缓存中对应的缓存行无效）；

第三：由于线程1的工作内存中缓存变量stop的缓存行无效，所以线程1再次读取变量stop的值时会去主存读取。

基于上面的描述，我们可能会问volatile这样的能力是不是能保证原子性了呢？答案是否定的，代码示例7

具体原因个人理解如下：

java语言的指令集是一门基于栈的指令集架构。也就是说它的数值计算是基于栈的。比如计算inc++,翻译成字节码就会变成：

0: iconst\_1

1: istore\_1

2: iinc 1, 1

0：的作用是把1放到栈顶

1：的作用是把刚才放到栈顶的1存入栈帧的局部变量表

2：的作用是对指令后面的1 ，1相加

由第0步可以看到，当指令序列将操作数存入栈顶之后就不再会从缓存中取数据了，那么缓存行无效也就没有什么影响了。

* Synchronized，用于标记一个方法或方法块，通过给对象上“锁”的方式，将自己的作用域变成一个临界区，只有获得锁的线程才可以进入临界区。每个java对象在内存中都有一个对应的监视器monitor，它用来存储“锁”标记，记录哪一个线程拥有这个对象的“锁”，又有哪些线程在竞争这个“锁”。锁，本质上是并发转串行，因此它天然就能解决原子性，可见性，有序性问题。代码示例8
* CAS与atomic包

Synchronized是一种独占锁，悲观锁，等待锁的线程处于BLOCKED状态，影响性能；锁的竞争会导致频繁的上下文切换和调度延时，开销较大；存在死锁的风险等等。基于这些问题，我们还有另外一个方案，那就是CAS(Compare And Swap)，其原理与我们常用的数据库乐观锁类似，即变量更新前检查当前值是否符合预期，如果符合则用新值替换当前值，否则就循环重试，直到成功。当下主流CPU直接在指令层面上支持了CAS指令，比如atomic底层调用的compareAndSwapInt方法就是这样一个native方法。因此，CAS的执行效率还是比较高的。

CAS在使用上还需要注意几点：

* 1. 通过版本号的方式，避免ABA问题
  2. 循环开销，冲突严重时过多地线程处于循环重试的状态，将增加CPU的负担
  3. 只能保证一个共享变量的原子性操作，如果想要多个变量同时保证原子性操作，可以考虑将这些变量放在一个对象中，然后使用AtomicReference类，这个类提供针对对象引用的原子性，从而保证对多个变量操作的原子性。代码示例9
* Lock自旋锁
  + Java提供了Lock接口以及其实现类ReentLock；ReadWriteLock接口以及其实现类ReentrantReadWriteLock
  + 与synchronized锁不同的是，线程在获取Lock锁的过程中不会被阻塞，而是通过循环不断的重试，直到当前持有该Lock锁的线程释放该锁
  + Synchronized是关键字，由编译器负责生成加锁和解锁操作，而ReentrantLock则是一个类，这个加锁和解锁的操作完全在程序员手中，因此在写代码时，调用了lock方法之后一定要记得调用unlock来解锁，最好放在finally块中

代码示例10

* Condition条件变量
  + Synchronized的同步机制要求所有线程等待同一对象的监视器“锁”标记。并且在通过wait/notify/notifyAll方法进行线程间通信时，只能随机或者全部且无序的唤醒这些线程，并没有办法“有选择”地决定要唤醒哪些线程，也无法避免“非公平锁”的问题
  + ReentrantLock允许开发者根据实际情况，创建多个条件变量，所有取得lock的线程可以根据不同的逻辑在对应的condition里面waiting，每个Condition对象拥有一个队列，用于存放处于waiting状态的线程
  + 这样的一种设计，同样可以让开发者根据实际情况，决定唤醒哪些condition内部waiting的线程，同时还能够实现公平锁。

代码示例11